I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Tree predictors for binary classification

Giorgia Carboni

Matriculation number: 24710A

2025

Statistical Methods for Machine Learning

Contents

1 Introduction

This paper presents a methodology for the implementation of **decision tree predictors** for binary classification, applied to the task of distinguishing edible from poisonous mushrooms using the *Secondary Mushroom Dataset*, which is a problem where accurate classification is crucial and where tree-based models offer advantages for handling diverse feature types. The implementation explores **three distinct criteria for leaf expansion**—Gini impurity, entropy (via Information Gain), and misclassification error—along with **three stopping criteria** designed to manage tree complexity and mitigate overfitting.

A significant component of this work involved meticulous preprocessing of the dataset, addressing numerous missing parameters to ensure data integrity. Subsequently, the decision tree structure was implemented, with careful consideration given to the selection of splitting mechanisms appropriate for the varying feature types encountered at each node, whether categorical or numerical.

This report will further detail the experimental setup, including a systematic **hyperparameter tuning** process utilizing a grid search approach to identify optimal model configurations. The resulting performance of the classifier will then be presented and analyzed.

The project features **Python 3.11** and some classic machine learning libraries like *sklearn*, *pandas* and *numpy*.

2 Methodology

This section details the methodology employed for the design and implementation of the decision tree classifier. It will cover the data preprocessing techniques applied, the structural definition of tree nodes, and the core algorithm part of the decision tree, including its construction process, feature splitting criteria, stopping mechanisms, and prediction logic. Finally, the training and hyperparameter tuning strategies will be showed.

The implementation is primarily organized across three Python scripts:

- `Node.py`: defines the structure for individual nodes within the tree.
- `DecisionTree.py`: encapsulates the main decision tree algorithm, including the constructor, the recursive tree-building process, methods for determining optimal splits, impurity calculations, application of stopping criteria and some helper functions.
- `training.py`: manages the overall workflow, handling dataset loading and extensive preprocessing, data partitioning, the execution of the hyperparameter tuning loop by instantiating and training DecisionTree models, and the evaluation of model performance.

2.1 Data Preprocessing

The dataset utilized for this project is the 'Secondary Mushroom Dataset', the main file (*secondary_ data.csv*) comprises 61,069 entries and 21 columns. This dataset aims to classify mushrooms as either edible ('e') or poisonous ('p'). An initial examination revealed that the target variable 'class' has a distribution of 33,888 poisonous samples vs 27,181 edible samples. The features are a mix of **numerical types** (3 columns, Dtype float64: *cap-diameter, stem-height* and *stem-width*) and object types (18 columns, Dtype object), the latter primarily representing **categorical** data. Several columns exhibited a significant number of **missing values** (e.g., cap-surface, gill-attachment, stem-root, veil-type), necessitating a series of preprocessing steps to prepare the data for the decision tree classifier.

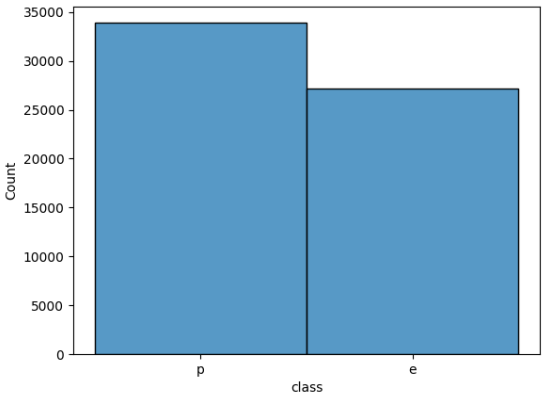

Figure 1: Class Distribution

Here is how everything was handled in the `training.py` script:

- **Numerical Features and Their Missing Values:** the function parse_numerical_range was applied to the columns containing numerical features. It was designed to convert string-formatted numerical features into floats, if the value is in list format, it takes the average between the two values (eg: [10, 20] becomes 15.0). Finally, it handles missing values and non-convertible values by returning *np.nan*. This function was designed to offer stability and to guarantee robustness in numerical data entries. The next problem it was tackled was handling with any remaining missing values. As a strategy, they were imputed using the **median** of their respective column (`df[col].fillna(df[col].median(), inplace=True)`)

- **Handling Categorical Features and Their Missing Values:** categorical columns were identified, excluding the *class* one.

  Missing values in these categorical columns were imputed using the **mode** (most frequent value) of each respective column (`df[col].fillna(mode[0], inplace=True)`), as this is the standardized method to handle these types of data.

- **Parsing and Standardizing Categorical Feature Formats:**

  Some categorical features contained multiple items in a single observation (for example: `[a, b, c]`). Being able to handle these types of features was crucial to ensure

a proper membership test during the splitting phase. In order to avoid problems when dealing with these types of data, two functions were employed:

- `parse_categorical_cell`: this function processes each cell in the categorical columns. If a list is encountered (identified by starting with '[' and ending with ']'), it parses the string into a list of individual category strings, otherwise it wraps the single string value into a list. Empty or NaN values are converted to empty lists.
- `ensure_list`: after imputation (which might fill NaNs with a single string mode), this function was applied to ensure that every entry in the processed categorical columns is indeed a list. This standardization is crucial for the `categorical_multi` logic within the decision tree, which expects list-like inputs for its membership tests.

- **Dropping Columns with High Percentage of Missing Values:** features which had more than 85% missing values were deemed as not influential for the final classification, as they led little to no information for the process, therefore they were dropped. Based on the count in `df.isnull().sum()`, the dropped columns were veil-type (94.8% missing), veil-color (87.9% missing) and spore-print-color (89.6% missing).

2.2 Node Implementation

The fundamental component of the decision tree is the node, which is implemented in the `Node.py` script. This class serves as a data structure to store information pertinent to each point in the tree, whether it's an internal decision node or a terminal leaf node. The design of the Node class follows these principles:

1. **Constructor**:

   The Node class is initialized using its constructor, `__init__(self, left=None, right=None, feature_idx=None, threshold=None, impurity=None, most_common_value=No`
   where:

   - `self.left` and `self.right`: These attributes are designed to store references to the left and right child nodes, respectively. As per the guidelines, these children are themselves instances of the Node class, forming the recursive structure of the tree.

   - `self.feature_idx`: Stores the index of the feature (column in the dataset). For example, if feature_idx is 4, it might refer to feature 'cap-color'. It would be None for a leaf node.

   - `self.threshold`: Holds the specific value (for numerical features) or category (for categorical features) against which the feature specified by feature_idx is compared.

   - `self.impurity`: This attribute stores the calculated impurity (e.g., Gini, en-

tropy) of the dataset subset that reaches this particular node.

- `self.most_common_value`: Stores the majority class label of the samples that fall into this node, it is computed by the function `_find_class`.

2. **Leaf Node Check (is_leaf()):**

   This method determines if a node is a terminal leaf node by returning True if both `self.left` and `self.right` children are None, and False otherwise.

The Node class itself does not directly contain the actual execution of the test logic- that is, taking a data point, accessing the relevant feature value using feature_idx, and comparing it against threshold to return a Boolean outcome or direct the traversal— as it was preferred to implement this part inside the `DecisionTree.py` script, to have more cohesion with the tree logic.

2.3 Decision Tree Implementation

This section represents the core of the whole project, as `DecisionTree.py` encapsulates the logic for constructing, splitting and training the tree. This implementation handles both numerical and multi-values categorical features.

2.3.1 Constructor `__init__`

The DecisionTree class is initialized via its constructor: `__init__(self, criterion, root=None, max_depth=8, min_samples_split=2, min_samples_per_leaf=1)`.

Here's a brief description of the input parameters:

1. `criterion`: A string specifying the impurity measure for the splits. This implementation supports 'gini' for **Gini impurity**, 'entropy' for **Information Gain**, and 'misclassification_error' as a third criterion.

2. `root`: An existing root node can be passed; otherwise, it's initialized to None and set during training.

3. `max_depth`: An integer defining the **maximum depth** the tree can grow. This acts as a stopping criterion to prevent overfitting.

4. `min_samples_split`: An integer representing the minimum number of samples required at a node to consider it for **splitting**. This is another stopping criterion.

5. `min_samples_per_leaf`: An integer specifying the **minimum number of samples that must reside in each child node** (leaf) after a split. If a split results in a leaf with fewer samples, the split is not performed, and the original node becomes a leaf.

2.3.2 Tree Building (`_build_tree`)

The tree structure is recursively built by `_build_tree(self, X, y, depth=0)`. At each node, this method first evaluates stopping criteria (detailed in Section 2.3.4). If no criteria are met, it invokes `_find_best_split()` to determine the optimal feature and threshold for partitioning the current data subset (X, y). The data is then divided, and `_build_tree()` is called recursively for the resulting left and right child subsets. An internal Node object stores the split parameters, while leaf nodes store the majority class prediction (derived from `_find_class()`).

2.3.3 Splitting Criteria

The `_find_best_split(self, X, y)` method is responsible for finding the **feature** and **threshold** that offer the best improvement for a split, according to the selected gain metric. It iterates through all features and their potential split points. The implementation supports three criteria:

1. **Gini Impurity**: The Gini impurity, $1 - \sum_k p_k^2$ (which is $2p(1-p)$ for binary cases), is calculated by `_gini(y)`. The goal is to maximize the reduction in Gini impurity.

2. **Information Gain (Entropy)**: Entropy, $H(p) = -\sum_k p_k \log_2 p_k$, calculated by `_entropy(y)`, measures data randomness. The split maximizing information gain, $IG = E_{\text{parent}} - \sum_{i \in \text{children}} w_i E_{\text{child}_i}$ (where $w_i = N_{\text{child}_i}/N_{\text{parent}}$), is chosen, as com-

puted by `_information_gain()`.

3. **Misclassification Error**: Calculated as $1 - \max_k p_k$ (or $\min\{p, 1-p\}$ for binary cases), by `_misclassification_error(y)`. While simple, this criterion can sometimes be less effective at selecting the most informative splits compared to Gini or entropy (as we will see in the results section).

Feature Handling in Splits:

- **Numerical Features**: Thresholds are generated as midpoints between unique sorted feature values. Data is split using `value <= threshold`.
- **Categorical Features (`categorical_multi`)**: For list-based categorical data, potential "thresholds" are individual unique categories. A sample is routed based on membership (e.g., `chosen_category` $\in$ `sample_feature_list`), implementing a test as described conceptually in.

2.3.4 Stopping Criteria

Tree growth is halted by several criteria within `_build_tree()` to prevent overfitting and ensure meaningful leaves:

- **Pre-splitting checks**:
  - The node is pure (all samples share the same class, determined by `_pure_node()`). Pure leaves do not contribute to training error.
  - Current `depth` reaches `self.max_depth`.
  - Number of samples (`len(y)`) is below `self.min_samples_split`.
- **Post-splitting checks** (after `_find_best_split()` but before recursion):
  - No effective split is found (`best_feature` is `None`).
  - A resulting child node would contain fewer samples than `self.min_samples_per_leaf`.

If any stopping criterion is met, the node becomes a leaf, predicting the majority class found by `_find_class(y)`.

2.3.5 Prediction (`predict`, `_traverse_tree`)

Predictions for new data `X` are generated by the `predict(X)` method, which iterates through samples, calling `_traverse_tree(x, node)` for each. This recursive helper method, `_traverse_tree()`, navigates the tree from the root for a given sample `x`. At each internal node, it applies the node's stored test (using `node.feature_idx` and `node.threshold`) to direct the sample to the appropriate child, correctly handling numerical and `categorical_multi` feature types. Upon reaching a leaf node (`node.is_leaf()` is `True`), the leaf's `most_common_value` is returned as the prediction for that sample.

2.4 Training Procedure (`fit`)

The primary method for training the tree is `fit(X, y)`. This function initiates the tree construction process by calling the recursive `_build_tree(X, y, depth=0)` method with the training features `X` and labels `y`. The root `Node` of the completed tree returned by `_build_tree()` is then stored as `self.root`, finalizing the model.

3 Experimental Setup

This final section discusses the design chosen to conduct the experiments and the results obtained.

The Secondary Mushroom dataset is composed by 61,069 samples:

- **Training Set**: 36,641 samples (approximately 601% of the data) were used to train the decision tree models.
- **Validation Set**: 12,214 samples (approximately 20% of the data) were used for hyperparameter tuning.
- **Test Set:** 12,214 samples (approximately 20% of the data) were held out for the final evaluation of the selected models

The features were processed as described in section 2. For reproducible data splits, a `random_state` was set; also, when the validation set was created from the training data, stratification by the *y_ train* target variable was applied to ensure that the proportion of classes (e.g., edible vs. poisonous) was similar in both the new training and validation subsets.

3.1 Hyperparameter Tuning

In `training.py`, a systematic **hyperparameter tuning** process was conducted to identify the optimal configuration for the decision tree classifier under different splitting criteria and using a dedicated validation set (and test set).

- **Splitting criteria explored**: Three splitting criteria were explored, as required: Gini Impurity ('gini'), Information Gain ('entropy') and Misclassification Error ('misclassification error').
- **Hyperparamters Tuned**: For each criterion mentioned above, a **grid search** was performed over the following ranges: max_depth: [5, 10, 12], min_samples_split: [2, 5], min_samples_per_leaf: [1, 2].

- **Evaluation Metric for Tuning**: The performance of each hyperparameter combination was assessed based on its **accuracy score on the validation set**.
- **Procedure**: For each of the three splitting criteria, the set of hyperparameters having the highest validation accuracy (`best_val_metric`) was selected as the optimal configuration for the chosen criterion. Subsequently, for each criterion, a model was trained on the combined training and validation sets (`X_train_val, y_train_val`) using the found optimal parameters.

3.2 Evaluation Metrics

The performance of the final, tuned models (one for each splitting criterion) was assessed on the unseen test set using the following metrics:

1. **Test Accuracy:** Portion of correctly classified samples in the test set.
2. **Training Error (0-1 Loss)**: Computed as $1 - accuracy$ on the combination training+validation set. This helps assessing overfitting.
3. **Confusion Matrix**: A table showing the counts of true positives, true negatives, false positives, and false negatives. Greatly helps visualizing how good the model performed on the data.
4. **Classification Report**: Provides class-specific precision, recall, and **F1-score**, giving a clearer picture of how well the model works for each category.

4 Results

This section presents the outcomes of the hyperparameter tuning process and the performance of the final decision tree models on the test set for each of the three splitting criteria, based on corrected code execution.

4.1 Gini Impurity Criterion

- **Hyperparameter Tuning Summary**:

- The grid search explored 12 combinations of hyperparameters for the Gini criterion.
- **Best Hyperparameters Found**: {'criterion': 'gini', 'max_depth': 12, 'min_samples_split': 2, 'min_samples_per_leaf': 2}.
- **Best Validation Accuracy**: 0.9456.
- *Observation*: For the Gini criterion, increasing `max_depth` from 5 (accuracy ≈0.7627) to 10 (accuracy ≈0.9288) and further to 12 (accuracy ≈0.9456) consistently improved validation accuracy. Within `max_depth=12`, a smaller `min_samples_split` of 2 generally performed marginally better or equal to `min_samples_split: 5`. The choice of `min_samples_per_leaf` (1 vs 2) showed minimal difference at higher depths, with `min_samples_per_leaf: 2` being part of the optimal configuration.

- **Final Model Performance (Gini)**:
  - **Test Accuracy**: 0.9483
  - **Training Error (0-1 Loss on train+val set)**: 0.0486
  - **Confusion Matrix (Test Set)**:

	Predicted Edible (0)	Predicted Poisonous (1)
Actual Edible (0)	5343 (TN)	70 (FP)
Actual Poisonous (1)	561 (FN)	6240 (TP)

  - **Classification Report Highlights (Test Set)**:
    - Class 0 (Edible): Precision=0.90, Recall=0.99, F1-score=0.94
    - Class 1 (Poisonous): Precision=0.99, Recall=0.92, F1-score=0.95

4.2 Entropy (Information Gain) Criterion

- **Hyperparameter Tuning Summary**:
  - The grid search explored 12 combinations for the Entropy criterion.
  - **Best Hyperparameters Found**: `{'criterion': 'entropy', 'max_depth': 12, 'min_samples_split': 2, 'min_samples_per_leaf': 1}`.
  - **Best Validation Accuracy**: 0.9027.
  - *Observation*: With the Entropy criterion, increasing `max_depth` significantly improved performance, from ≈0.7492 at `max_depth=5` to ≈0.8657 at `max_depth=10`, and further to ≈0.9027 at `max_depth=12`. At `max_depth=12`, all tested combinations of `min_samples_split` (2 or 5) and `min_samples_per_leaf` (1 or 2) yielded the same peak validation accuracy, indicating robustness at this depth for these settings.
- **Final Model Performance (Entropy)**:
  - **Test Accuracy**: 0.8984
  - **Training Error (0-1 Loss on train+val set)**: 0.0917
  - **Confusion Matrix (Test Set)**:

	Predicted Edible (0)	Predicted Poisonous (1)
Actual Edible (0)	5061 (TN)	352 (FP)
Actual Poisonous (1)	889 (FN)	5912 (TP)

  - **Classification Report Highlights (Test Set)**:
    - Class 0 (Edible): Precision=0.85, Recall=0.93, F1-score=0.89
    - Class 1 (Poisonous): Precision=0.94, Recall=0.87, F1-score=0.91

4.3 Misclassification Error Criterion

- **Hyperparameter Tuning Summary**:
  - The grid search explored 12 combinations for the Misclassification Error criterion.
  - **Best Hyperparameters Found**: `{'criterion': 'misclassification_error', 'max_depth': 12, 'min_samples_split': 2, 'min_samples_per_leaf': 1}`.
  - **Best Validation Accuracy**: 0.8848.
  - *Observation*: For the Misclassification Error criterion, increasing `max_depth` from 5 (accuracy ≈0.7993) to 10 (accuracy ≈0.8776) and to 12 (accuracy ≈0.8848) showed consistent improvements. At `max_depth=12`, `min_samples_split: 2, min_samples_per_leaf: 1` and `min_samples_split: 5, min_samples_per_leaf: 1` both achieved the peak validation accuracy.
- **Final Model Performance (Misclassification Error)**:
  - **Test Accuracy**: 0.8652
  - **Training Error (0-1 Loss on train+val set)**: 0.1255
  - **Confusion Matrix (Test Set)**:

	Predicted Edible (0)	Predicted Poisonous (1)
Actual Edible (0)	4522 (TN)	891 (FP)
Actual Poisonous (1)	755 (FN)	6046 (TP)

  - **Classification Report Highlights (Test Set)**:
    - Class 0 (Edible): Precision=0.86, Recall=0.84, F1-score=0.85
    - Class 1 (Poisonous): Precision=0.87, Recall=0.89, F1-score=0.88

4.4 Summary Comparison of Criteria

Table 1: Summary Comparison of Splitting Criteria Performance (Corrected Runs)

Splitting Criterion	Best Validation Accuracy	Final Test Accuracy	Training Error (0-1 Loss)
Gini Impurity	0.9456	**0.9483**	0.0486
Entropy	0.9027	0.8984	0.0917
Misclassification Error	0.8848	0.8652	0.1255

The Gini impurity criterion yielded the highest test accuracy.

5 Discussion

The results obtained from the implementation and evaluation of the decision tree classifiers, following corrected hyperparameter tuning, provide several key insights into their performance and the effectiveness of the adopted methodology.

5.1 Overall Model Performance and Choice of Criterion

The decision tree classifier using the **Gini impurity criterion** demonstrated superior performance, achieving an excellent **test accuracy of 0.9483**. This was notably higher than the models using Entropy (0.8984) and Misclassification Error (0.8652). The Gini criterion's tendency to isolate the majority class in one branch, coupled with effective hyperparameter choices, likely contributed to its robust performance.

5.2 Discussion on Overfitting and Underfitting

A key objective was to build a model that generalizes well without significant overfitting. The results from the corrected runs are as follows:

- For the **Gini model** (Optimal params: `max_depth: 12, min_samples_split: 2, min_samples_per_leaf: 2`):
  - Training Error (0-1 Loss on train+val set): 0.0486 (4.86%)
  - Test Error (1 - Test Accuracy): $1 - 0.9483 = 0.0517$ (5.17%)

The difference between the training error and the test error is $0.0517 - 0.0486 = 0.0031$ (0.31%). This very small difference strongly suggests that the model **did not significantly overfit** the training data and generalized exceptionally well to the unseen test set.

- For the **Entropy model** (Optimal params: `max_depth: 12, min_samples_split: 2, min_samples_per_leaf: 1`):
  - Training Error: 0.0917 (9.17%)
  - Test Error: $1 - 0.8984 = 0.1016$ (10.16%)

  The gap is $0.1016 - 0.0917 = 0.0099$ (0.99%), also indicating good generalization.

- For the **Misclassification Error model** (Optimal params: `max_depth: 12, min_samples_split: 2, min_samples_per_leaf: 1`):
  - Training Error: 0.1255 (12.55%)
  - Test Error: $1 - 0.8652 = 0.1348$ (13.48%)

  The gap is $0.1348 - 0.1255 = 0.0093$ (0.93%), also indicating good generalization.

The successful management of overfitting for all criteria can be attributed to the **sound hyperparameter tuning procedure** and the effective use of stopping criteria. For the best-performing Gini model, the optimal parameters found were `max_depth: 12`, `min_samples_split: 2`, and `min_samples_per_leaf: 2`. This configuration allowed the tree to grow to a sufficient depth to capture complex patterns while the `min_samples_split` and `min_samples_per_leaf` parameters prevented it from creating overly specific splits on too few samples, thus avoiding memorization of noise in the training data. This demonstrates that the chosen stopping criteria and tuning strategy were effective in finding an excellent balance between model complexity and predictive power.

Conclusion

This project successfully demonstrated that a from-scratch implementation of a decision tree binary classifier for mushroom edibility can achieve high accuracy.

The classifier utilizing the Gini impurity criterion achieved the highest performance, with an outstanding **test accuracy of 0.9483**. A key outcome was the model's **strong generalization capability**, evidenced by a minimal 0.31% difference between training (4.86% 0-1 Loss) and test error (5.17%), indicating that overfitting was effectively controlled through rigorous hyperparameter tuning. The optimal Gini configuration was found with parameters including `max_depth: 12`, `min_samples_split: 2`, and `min_samples_per_leaf: 2`.

Despite this strong performance, the Gini model's 561 False Negatives (misclassifying poisonous mushrooms as edible) highlight a critical area requiring careful consideration for safety-critical applications.

Future work could build upon these findings by:

1. Incorporating cost-sensitive learning to penalize high-risk misclassifications more heavily.
2. Exploring ensemble approaches such as Random Forests, as proposed in the project guidelines, to improve robustness and accuracy.
3. Performing further feature analysis or engineering to improve class separability.

In summary, this project demonstrated that excellent classification accuracy is attainable by the critical use of sound methodology and careful hyperparameter tuning.

A Graphs

Confusion matrices of the different criteria used (gini, information gain and misclassification error).

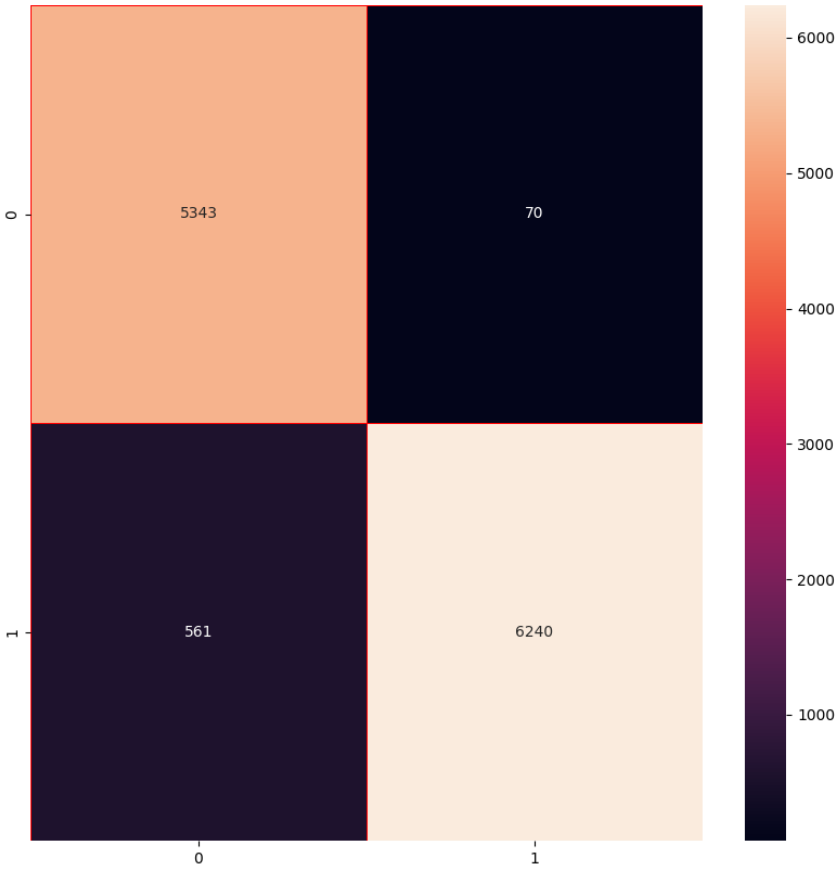

Figure 2: Model using Gini

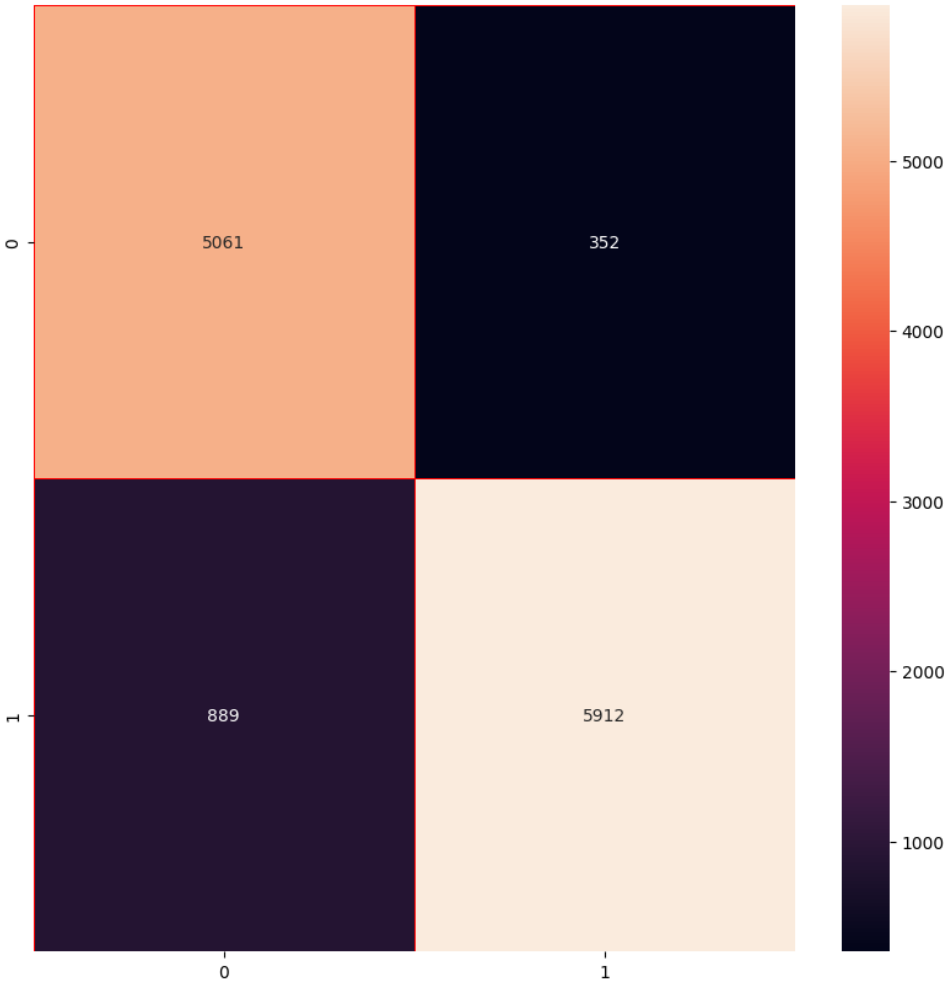

Figure 3: Model using Entropy and Information Gain

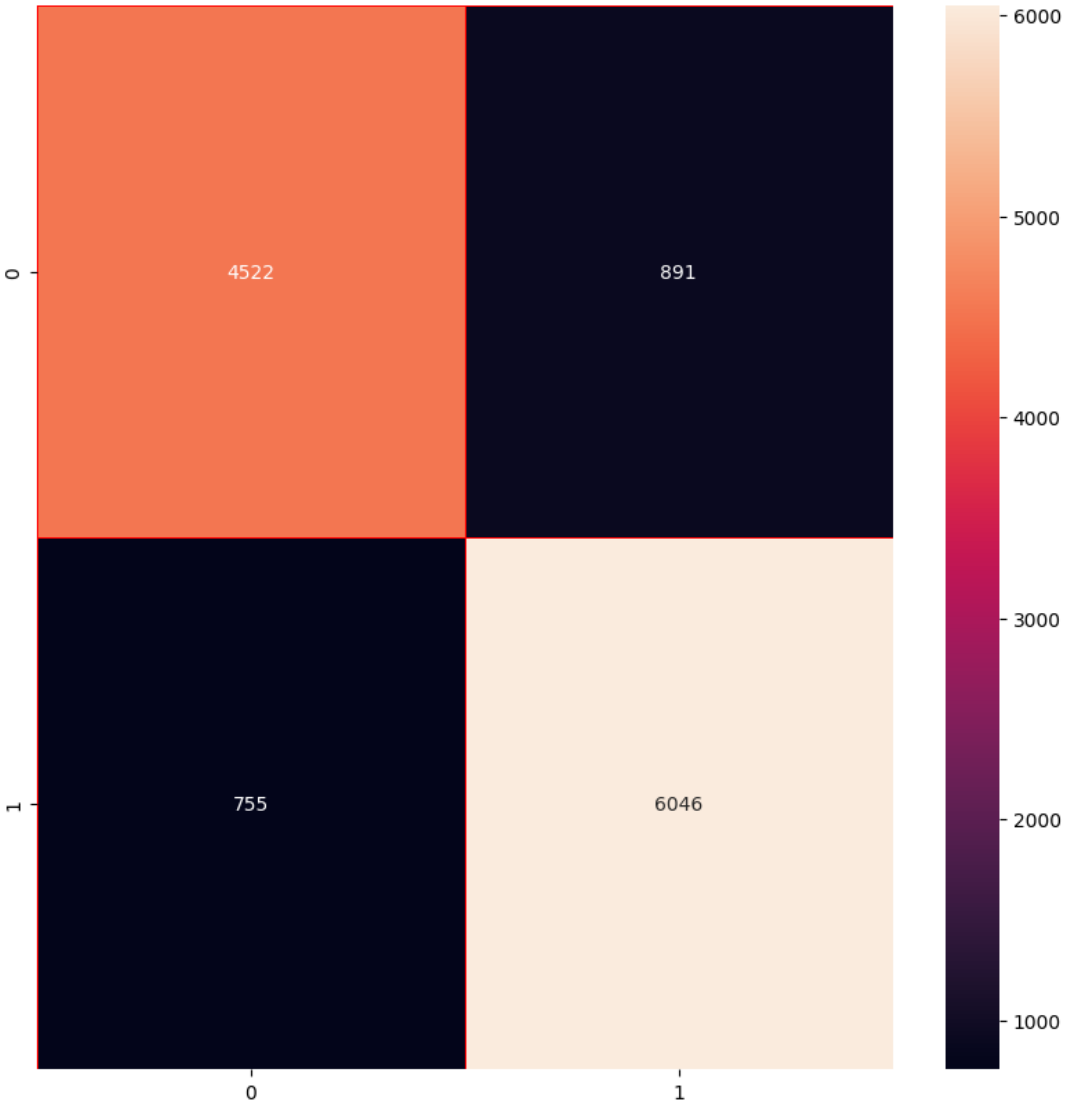

Figure 4: Model using Misclassification Error